# 1.6 — Uninitialized variables and undefined behavior

ALEX[1] FEBRUARY 15, 2024

## Uninitialized variables

Unlike some programming languages, C/C++ does not automatically initialize most variables to a given value (such as zero). When a variable that is not initialized is given a memory address to use to store data, the default value of that variable is whatever (garbage) value happens to already be in that memory address! A variable that has not been given a known value (through initialization or assignment) is called an **uninitialized variable**.

### Nomenclature

Many readers expect the terms "initialized" and "uninitialized" to be strict opposites, but they aren't quite! In common language, "initialized" means the object was provided with an initial value at the point of definition. "Uninitialized" means the object has not been given a known value yet (through any means, including assignment). Therefore, an object that is not initialized but is then assigned a value is no longer *uninitialized* (because it has been given a known value).

To recap:

- Initialized = The object is given a known value at the point of definition.
- Assignment = The object is given a known value beyond the point of definition.
- Uninitialized = The object has not been given a known value yet.

Relatedly, consider this variable definition:

```
1 | int x;
```

In lesson 1.4 -- Variable assignment and initialization (https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/)[2], we noted that when no initializer is provided, the variable is default-initialized. In most cases (such as this one), default-initialization performs no actual initialization. Thus we'd say `x` is uninitialized. We're focused on the outcome (the object has not been given a known value), not the process.

### As an aside...

This lack of initialization is a performance optimization inherited from C, back when computers were slow. Imagine a case where you were going to read in 100,000 values from a file. In such case, you might create 100,000 variables, then fill them with data from the file.

If C++ initialized all of those variables with default values upon creation, this would result in 100,000 initializations (which would be slow), and for little benefit (since you're overwriting those values anyway).

For now, you should always initialize your variables because the cost of doing so is minuscule compared to the benefit. Once you are more comfortable with the language, there may be certain cases where you omit the initialization for optimization purposes. But this should always be done selectively and intentionally.

Using the values of uninitialized variables can lead to unexpected results. Consider the following short program:

```
#include <iostream>

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized because we haven't given it a
value

    // print the value of x to the screen
    std::cout << x << '\n'; // who knows what we'll get, because x is
uninitialized

    return 0;
}
```

In this case, the computer will assign some unused memory to *x*. It will then send the value residing in that memory location to *std::cout*, which will print the value (interpreted as an integer). But what value will it print? The answer is "who knows!", and the answer may (or may not) change every time you run the program. When the author ran this program in Visual Studio, *std::cout* printed the value `7177728` one time, and `5277592` the next. Feel free to compile and run the program yourself (your computer won't explode).

## Warning

Some compilers, such as Visual Studio, *will* initialize the contents of memory to some preset value when you're using a debug build configuration. This will not happen when using a release build configuration. Therefore, if you want to run the above program yourself, make sure you're using a *release build configuration* (see lesson 0.9 -- Configuring your compiler: Build configurations (https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-build-configurations/)[3] for a reminder on how to do that). For example, if you run the above program in a Visual Studio debug configuration, it will consistently print -858993460, because that's the value (interpreted as an integer) that Visual Studio initializes memory with in debug configurations.

Most modern compilers will attempt to detect if a variable is being used without being given a value. If they are able to detect this, they will generally issue a compile-time warning or error. For example, compiling the above program on Visual Studio produced the following warning:

```
c:\VCprojects\test\test.cpp(11) : warning C4700: uninitialized local variable 'x' used
```

If your compiler won't let you compile and run the above program (e.g. because it treats the issue as an error), here is a possible solution to get around this issue:

```
#include <iostream>

void doNothing(int&) // Don't worry about what & is for now, we're just using it to trick the compiler into thinking variable x is used
{
}

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized

    doNothing(x); // make the compiler think we're assigning a value to this variable

    // print the value of x to the screen (who knows what we'll get, because x is uninitialized)
    std::cout << x << '\n';

    return 0;
}
```

Using uninitialized variables is one of the most common mistakes that novice programmers make, and unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized variable happened to get assigned to a spot of memory that had a reasonable value in it, like 0).

This is the primary reason for the "always initialize your variables" best practice.

---

## Undefined behavior

Using the value from an uninitialized variable is our first example of undefined behavior. **Undefined behavior** (often abbreviated **UB**) is the result of executing code whose behavior is not well-defined by the C++ language. In this case, the C++ language doesn't have any rules determining what happens if you use the value of a variable that has not been given a known value. Consequently, if you actually do this, undefined behavior will result.

Code implementing undefined behavior may exhibit *any* of the following symptoms:

- Your program produces different results every time it is run.
- Your program consistently produces the same incorrect result.
- Your program behaves inconsistently (sometimes produces the correct result, sometimes not).
- Your program seems like it's working but produces incorrect results later in the program.

- Your program crashes, either immediately or later.
- Your program works on some compilers but not others.
- Your program works until you change some other seemingly unrelated code.

Or, your code may actually produce the correct behavior anyway.

### Author's note

Undefined behavior is like a box of chocolates. You never know what you're going to get!

C++ contains many cases that can result in undefined behavior if you're not careful. We'll point these out in future lessons whenever we encounter them. Take note of where these cases are and make sure you avoid them.

### Rule

Take care to avoid all situations that result in undefined behavior, such as using uninitialized variables.

### Author's note

One of the most common types of comment we get from readers says, "You said I couldn't do X, but I did it anyway and my program works! Why?".

There are two common answers. The most common answer is that your program is actually exhibiting undefined behavior, but that undefined behavior just happens to be producing the result you wanted anyway... for now. Tomorrow (or on another compiler or machine) it might not.

Alternatively, sometimes compiler authors take liberties with the language requirements when those requirements may be more restrictive than needed. For example, the standard may say, "you must do X before Y", but a compiler author may feel that's unnecessary, and make Y work even if you don't do X first. This shouldn't affect the operation of correctly written programs, but may cause incorrectly written programs to work anyway. So an alternate answer to the above question is that your compiler may simply be not following the standard! It happens. You can avoid much of this by making sure you've turned compiler extensions off, as described in lesson 0.10 -- Configuring your compiler: Compiler extensions (https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-compiler-extensions/)[4].

## Implementation-defined behavior and unspecified behavior

**Implementation-defined behavior** means the behavior of some syntax is left up to the implementation (the compiler) to define. Such behaviors must be consistent and documented, but different compilers may produce different results.

Let's look at a simple example of implementation-defined behavior:

```
#include <iostream>

int main()
{
    std::cout << sizeof(int) << '\n'; // print how many bytes of memory an
int value takes

    return 0;
}
```

On most platforms, this will produce `4`, but on others it may produce `2`.

### Related content

We discuss `sizeof()` in lesson 4.3 -- Object sizes and the sizeof operator (https://www.learncpp.com/cpp-tutorial/object-sizes-and-the-sizeof-operator/)[5].

**Unspecified behavior** is almost identical to implementation-defined behavior in that the behavior is left up to the implementation, but the implementation is not required to document the behavior.

We generally want to avoid implementation-defined and unspecified behavior, as it means our program may not work as expected if compiled on a different compiler (or even on the same compiler if we change project settings that affect how the implementation behaves!)

### Best practice

Avoid implementation-defined and unspecified behavior whenever possible, as they may cause your program to malfunction on other implementations.

### Related content

We show examples of unspecified behavior in lesson 6.1 -- Operator precedence and associativity (https://www.learncpp.com/cpp-tutorial/operator-precedence-and-associativity/#unspecified)[6].

---

## Quiz time

#### Question #1

What is an uninitialized variable? Why should you avoid using them?

Hide Solution (javascript:void(0))[7]

> An uninitialized variable is a variable that has not been given a value by the program (generally through initialization or assignment). Using the value stored in an uninitialized variable will result in undefined behavior.

---

#### Question #2

What is undefined behavior, and what can happen if you do something that exhibits undefined behavior?

Show Solution (javascript:void(0))[7]

---

## Next lesson

1.7 Keywords and naming identifiers

8

## Back to table of contents

9

## Previous lesson

1.5 Introduction to iostream: cout, cin, and endl

10

11

---

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[13] are connected to your provided email address.

230 COMMENTS Newest

**Newchallenger147**
March 9, 2024 3:47 am

What do you mean "there may be certain cases where you omit the initialization for optimization purposes." from what I understand Instead of using x{}, instead I should x; ?

0 Reply

**Alex** Author
Reply to Newchallenger147 [14] March 11, 2024 5:22 pm

Normally we initialize our objects even if we then immediately overwrite their values (e.g. by getting input from the user, or reading in data from a file). This is because our default is to favor safety and predictability over speed.

Now imagine a case where we're going to read in 1GB of data from disk. Initializing the object holding this data before reading in the data from disk will probably be expensive due to the immense size. In such cases, we may opt to skip initializing and just go straight to reading from disk.

This should only be done in particular circumstances, and when it actually makes a notable difference.

5 Reply

**beginner123**
March 4, 2024 10:08 am

Please correct me if I'm wrong but:

int x; is \\uninitalized.

However, during the program, I ask the user

cin >> x;

would this turn that int x uninitialized to an initialized integer?

0 Reply

**Alex** Author
Reply to beginner123 [15] March 6, 2024 2:55 pm

Yes.

1 Reply

**Likss**

Reply to Alex [16] April 3, 2024 10:33 am

What will be the value of the variable till the user has not given the input

0 Reply

**xeno**

Reply to Likss [17] April 4, 2024 8:24 am

Unlike some programming languages, C/C++ does not automatically initialize most variables to a given value (such as zero). When a variable that is not initialized is given a memory address to use to store data, the default value of that variable is whatever (garbage) value happens to already be in that memory address!

0 Reply

**cake**

February 27, 2024 8:06 am

Undefined behaviour in most common von-neumann machine:

Give your toddler a s*haped chocolate. I am sure they're gonna show all the above UBs.

0 Reply

**Demon**

February 22, 2024 9:35 pm

Hello, been going through this site and it seems quite informative and ty for the work :D
Though I have an issue, I try to understand a lot of deep concepts at once which tends to me going into deep rabbit holes I don't understand XD

And being a simpleton, Undefined Behavior, Implementation Defined, Unspecified Behavior I can't really understand it well at all.
Mostly its hard finding out what is or isn't defined amongst all of these. Also I don't understand the differences with Implementation Defined things and Unspecified. Might be linked to not knowing what causes them, different compilers, different computers etc.

Do you have tips?
Seen various sources so far but I still can't get an idea of it.
I feel like at some point I need to skip understanding this for now so I can continue learning.

0 Reply

**cake**

Reply to Demon [18] February 27, 2024 8:30 am

When we say implementation-defined behaviour, it means the behaviour shown by the code (or a code block) based on the design of the compiler, that is, how the compiler works for the given syntax. From the above example, "std::cout << sizeof(int);" will return the output as '4' for me, while others may get output as '2' or even '8'. Well these things are well documented for every respective compiler and their versions.

When we say unspecified behaviour, it means you also can't tell what the output will be until you run the same program in other compiler, that's why you should avoid it.

That's why we should avoid the above two, as they are compiler based designing of the program. You should focus it to be based on architecture rather than compiler based, or god knows when your program is going to give correct results :D

Undefined behaviour means the behaviour shown due to the uninitialized variable(s). FYI, there's always some garbage (random values) laid in the cells (basic block of storage) of the RAM (main memory). When we define a variable, it gets a memory block allocated to itself. Now that the variable is not initialized, nor assigned to any value, it grabs the rubbish available in the memory block allocated to itself. In this case if you're using this variable you'll be getting different output (maybe) everytime. Also sometimes the program may work, sometimes it won't or it's going to crash anytime.

Note: I wrote whatever I understood after going through this tutorial, so if you find anything wrong please do correct it. :)

*Last edited 1 month ago by cake*

2 Reply

**Eduard**

February 17, 2024 8:53 am

"This lack of initialization is a performance optimization inherited from C, back when computers were slow. Imagine a case where you were going to read in 100,000 values from a file. In such case, you might create 100,000 variables, then fill them with data from the file."

This is still not an excuse.
There is really no problem with initializing variables later then when they are declared, the problem is the compilers not performing proper checks.

Yet another of the many problems that do not occur in a language like Rust.
In Rust you can perfectly write
let x;
and assign it a value later. If you try to read from it before assigning, your program simply won't compile.
The the compiler will give you a clear message.

But even if you give variables an initial value, a compiler can easily detect that and optimize it away, because you didn't use it.

Let's just call the design of C and C++ with these things a bad really choice.
That's the only fair answer.

Most bugs in C/C++ programs are bugs related to invalid reads/writes in memory. And they open many ways for security vulnerabilities.

*Last edited 1 month ago by Eduard*

0 Reply

**Flicker**

Reply to Eduard [19] February 18, 2024 11:30 pm

Is this some kind of rust ad? Do you not have anything better to do.

0 Reply

**Noob**

Reply to Flicker [20] February 20, 2024 3:45 pm

yeah its an ad

0 Reply

**Eduard**

Reply to Noob [21] February 26, 2024 2:17 am

It's not. This is called criticism. :)
I am not saying that you shouldn't learn C or C++ either (it's actually a great idea), but I am just sharing my thoughts.
I simply think tutorials like these should come up with better quality of information and give more actual reasons for things.

0 Reply

**Noob**

Reply to Eduard [22] February 26, 2024 4:36 am

(I apologize if the following sounds ruder than i meant it) I only said that because I saw you had another statement about rust. if that wasnt your intent i apologize. I appreciate you being polite as well.

1 Reply

**IceFloe**
February 15, 2024 5:58 pm

are there any cases where undefined behavior is used intentionally? If so, in what cases?

1 Reply

**Alex** Author
Reply to IceFloe [23] February 17, 2024 2:07 pm

Not to my knowledge.

0 Reply

**Eduard**
Reply to IceFloe [23] February 17, 2024 9:07 am

The only thing I can think of is some cases with integer overflow, for which C/C++ compilers can do some optimizations.
But still, the damage done by UB is generally much bigger and rarely worth allowing it.

This is why the Rust language eliminates UB in safe code, very different from C/C++.

Even if you allow UB in a language, the developer still needs to introduce additional checks to prevent it - which also come with a performance overhead. Net result is zero win in most cases.

*Last edited 1 month ago by Eduard*

0 Reply

**Noob**
Reply to Eduard [24] February 20, 2024 3:44 pm

Yo theres a ad guy here

0 Reply

**Eduard**
Reply to Noob [25] February 26, 2024 2:22 am

Please read my other reply above.
I am totally fine if you choose to disagree, but I am **not** advertising.

0 Reply

**Kin**
Reply to Eduard [26] April 9, 2024 3:05 am

Yet your comments read like ads. It's fine to criticize explanations and reasoning for things (as long as it's done in a constructive manner), it may help the tutorial. I consider it a bit nit-picky for a statement in a tutorial where most readers are just trying to get the basics of the language down, but that's just my opinion and it's absolutely fine to consider it more relevant than I do.

There is no point in referencing another language in every post that is not made in reply to someone calling your post ad-messages. Sure, Rust deals with certain things in a way that removes issues which may arise in C/C++, but that is pretty irrelevant to a tutorial for C++.

0 Reply

**Alexander**
February 15, 2024 10:10 am

Visual Studio doesn't allow the program to be compiled if it contains an uninitialized variable. It gives me this error: **uninitialized local variable 'x' used**
Is this a problem?

0 Reply

**Qwerty**
Reply to Alexander [27] February 17, 2024 8:07 am

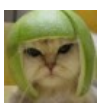
no, that isn't a problem. if you would like to make it think that the variable is initialized but actually isn't (so that you can run it), you can use the snippet of code below that part.

0 Reply

**HiMom**
February 11, 2024 12:18 pm

`On most platforms, this will produce 4, but on others it may produce 2.` I think most outputs will be 8 for an int

0 Reply



**Alex** Author
Reply to HiMom [28] February 11, 2024 11:52 pm

`int` is not 8 bytes on any of the standard architectures.

0 Reply

**Diego**
February 6, 2024 5:48 pm

Really appreciate your effort Alex, you are crafting some future programmers!

3 Reply

**ftKhateeb**
February 3, 2024 12:06 am

in Lesson 1.4 in Question 3

int x; // x is called default value initialized

in this lesson

int x; // x is called noninitialized

Which one is correct ?

0 Reply

**Alex** Author
Reply to ftKhateeb [29] February 3, 2024 11:45 am

Both.

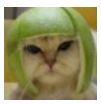
Technically, `x` is default initialized. However, for fundamental types such as `int`, default initialization does not initialize the variable. So conventionally speaking, the variable is uninitialized.

0 Reply

**naiven**
January 27, 2024 10:42 am

hey alex, i wanted to ask do you maybe have a discord server to ask for instructions with self written code exercises ? or a forum ( i read you didn't and only knew community such as stackoverflow or subreddit somewhere though)? Like i'm practicing with the tuto and some beginner python knowledge and wanted to get someone maybe to help if there's an error i don't understand.

0 Reply



**Alex** Author
Reply to naiven [30] January 31, 2024 5:59 pm

Nope. Just this site and these comments.

0 Reply

**Krishnakumar**
January 20, 2024 10:43 pm

gcc 12.0 and clang 8.0 onwards have a flag `-ftrivial-auto-var-init`. One of the options that this can be set is to zero. i.e. `-ftrivial-auto-var-init=zero`. There is also an accompanying flag `-Wtrivial-auto-var-init`.

Quoting from the GCC online manual:

"-ftrivial-auto-var-init=choice

Initialize automatic variables with either a pattern or with zeroes to increase the security and predictability of a program by preventing uninitialized memory disclosure and use. GCC still considers an automatic variable that doesn't have an explicit initializer as uninitialized, -Wuninitialized and -Wanalyzer-use-of-uninitialized-value will still report warning messages on such automatic variables and the compiler will perform optimization as if the variable were uninitialized. With this option, GCC will also initialize any padding of automatic variables that have structure or union types to zeroes. However, the current implementation cannot initialize automatic variables that are declared between the controlling expression and the first case of a switch statement. Using -Wtrivial-auto-var-init to report all such cases.

The three values of choice are:

'uninitialized' doesn't initialize any automatic variables. This is C and C++'s default.
'pattern' Initialize automatic variables with values which will likely transform logic bugs into crashes down the line, are easily recognized in a crash dump and without being values that programmers can rely on for useful program semantics. The current value is byte-repeatable pattern with byte "0xFE". The values used for pattern initialization might be changed in the future.
'zero' Initialize automatic variables with zeroes.
"

---

"-Wtrivial-auto-var-init

Warn when -ftrivial-auto-var-init cannot initialize the automatic variable. A common situation is an automatic variable that is declared between the controlling expression and the first case label of a switch statement."

0 Reply

**Tortik**
January 10, 2024 2:45 pm

Thanks, Alex!
I have started learning c++ frim your tutorials few days ago.
I like how you manage to explain complex topics using simplistic terminology and also by providing alternatives.
I also like the design of the website and the colour scheme.
Keep going! :)

4 Reply

**Simon**
January 5, 2024 11:19 am

if undefined code is like a box of chocolates, why doesn't it ever taste good?

1 Reply

**Rosy**
Reply to Simon [31] January 27, 2024 6:03 am

It's all that horrible strawberry chocolate

3 Reply

**Andreas**
December 22, 2023 1:47 am

Okay, there's one thing I need to ask: I do understand that from a performance standpoint this makes sense. But doesn't this also mean that a program could misuse that and read potentially sensitive information from other programs? The random value of an uninitialized variable could be the part of a password and stuff like that, right?

0 Reply

**Alex** Author
Reply to Andreas [32] December 22, 2023 2:53 pm

In theory, yes. But modern OSes will generally zero-out memory that was previously allocated to other processes to prevent this from happening.

1 Reply

**Linuxer**
December 14, 2023 12:53 pm

This is the best written C++ guide on internet!!

5 Reply

**AbeerOrTwo**
December 13, 2023 8:26 pm

Helpful helpful

1 Reply

**Makis**
December 6, 2023 11:18 am

It does not say some random integer when I run this program It just says 0.

*Last edited 4 months ago by Makis*

0 Reply

**Alex** Author
Reply to Makis [33] December 7, 2023 4:13 pm

The lesson says:

> But what value will it print? The answer is "who knows!", and the answer may (or may not) change every time you run the program.

Consistently printing 0 is an expected possible outcome.

1 Reply

**k3d**
Reply to Makis [33] December 6, 2023 6:04 pm

Yes, I am using VScode and having the same outcome.

1 Reply

**Kaloyan**
Reply to k3d [34] December 29, 2023 12:56 pm

Same with me too. I'm also using VS code and exhibiting the same behavior of only zeros even with arguments "-pedantic-errors", and "-O2", "-DNDEBUG", (for switching to release built mode) in the `tasks.json` file.

0 Reply

**Raghvan**
November 13, 2023 1:25 pm

```
#include <iostream>

void doNothing(int&) // Don't worry about what & is for now, we're just
using it to trick the compiler into thinking variable x is used
{
}

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized

    doNothing(x); // make the compiler think we're assigning a value to this
variable

    // print the value of x to the screen (who knows what we'll get, because
x is uninitialized)
    std::cout << x << '\n';

    return 0;
}
```

in this program on debug solution configuration I'm getting an error
`Error LNK1168 cannot open C:\Users\raghv\source\repos\Variable_assignment\x64\Debug\Variable_assignment.exe for writing Variable_assignment C:\Users\raghv\source\repos\Variable_assignment\LINK 1`

when i searched it online it gave me a linker error kind of thing...
So i changed it to release config and then it run successfully but always prints `0` as result...i experienced it in programs of previous topics also

Is my VS set to any kind of default initialization????helpppppp plzzzz...

1 Reply

**hihh**
Reply to Raghvan [35] December 6, 2023 9:41 am

"**Using uninitialized variables is one of the most common mistakes that novice programmers make, and unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized variable happened to get assigned to a spot of memory that had a reasonable value in it, like 0 ).

This is the primary reason for the “always initialize your variables” best practice.**"

1 Reply

**Felix**
October 6, 2023 3:40 am

Hey Alex, in the Implementation-defined behaviors section, you explained that some syntax depend on the compiler to define the results or behavior of that syntax like sizeof(), but is it really the compiler? or the architecture of the processor that defines those behaviors, i feel its the processor that determines that or am I wrong ?

*Last edited 6 months ago by Felix*

0 Reply

**Alex** Author
Reply to Felix [36] October 6, 2023 11:10 am

The compiler defines those behaviors, based on whatever makes the most sense for the platform.

1 Reply

# Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/
3. https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-build-configurations/
4. https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-compiler-extensions/
5. https://www.learncpp.com/cpp-tutorial/object-sizes-and-the-sizeof-operator/
6. https://www.learncpp.com/cpp-tutorial/operator-precedence-and-associativity/#unspecified
7. javascript:void(0)
8. https://www.learncpp.com/cpp-tutorial/keywords-and-naming-identifiers/
9. https://www.learncpp.com/
10. https://www.learncpp.com/cpp-tutorial/introduction-to-iostream-cout-cin-and-endl/
11. https://www.learncpp.com/uninitialized-variables-and-undefined-behavior/
12. https://www.learncpp.com/cpp-tutorial/introduction-to-literals-and-operators/
13. https://gravatar.com/
14. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-594470
15. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-594259
16. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-594369

17. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-595404
18. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593939
19. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593745
20. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593807
21. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593867
22. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-594047
23. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593677
24. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593746
25. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593866
26. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-594048
27. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593661
28. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593539
29. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-593181
30. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-592960
31. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-591790
32. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-591261
33. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-590608
34. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-590619
35. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-589736
36. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/#comment-588294